23 | 钨丝计划：Tungsten给开发者带来了哪些福报？

2021-05-05 吴磊

Spark性能调优实战 进入课程 >

讲述：吴磊

时长 21:42 大小 19.88M

你好，我是吴磊。

通过前两讲的学习，我们知道在 Spark SQL 这颗智能大脑中，“左脑”Catalyst 优化器负责把查询语句最终转换成可执行的 Physical Plan。但是，把 Physical Plan 直接丢给 Spark 去执行并不是最优的选择，最优的选择是把它交给“右脑”Tungsten 再做一轮优化。

Tungsten 又叫钨丝计划，它主要围绕内核引擎做了两方面的改进：数据结构设计和全阶段代码生成（WSCG，Whole Stage Code Generation）。

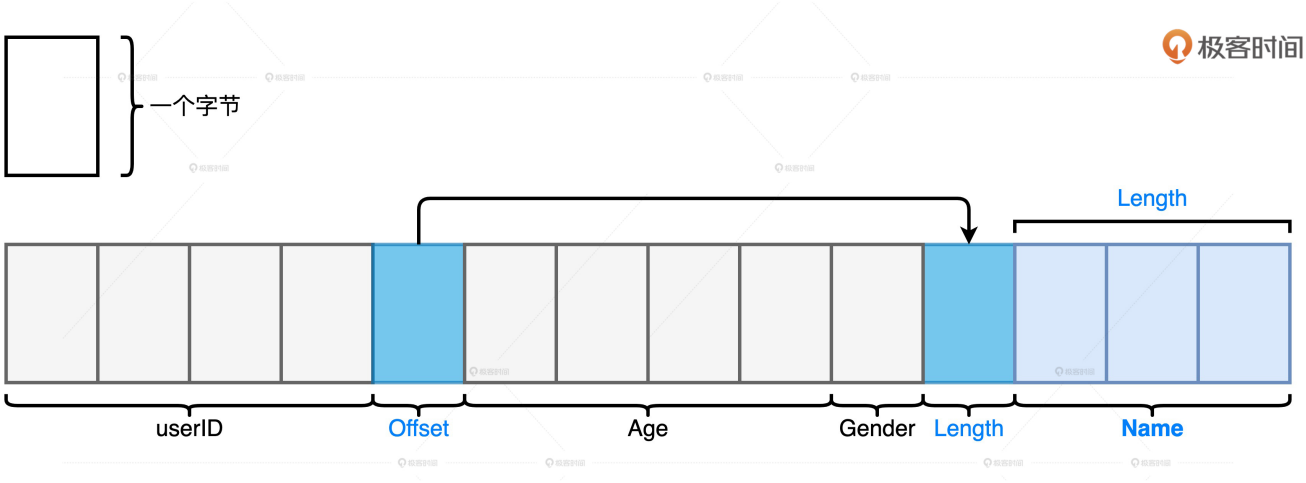
今天这一讲，我们就来说说 Tungsten 的设计初衷是什么，它的两方面改进到底解决了哪些问题，以及它给开发者到底带来了哪些性能红利。

Tungsten 在数据结构方面的设计

相比 Spark Core，Tungsten 在数据结构方面做了两个比较大的改进，一个是紧凑的二进制格式 Unsafe Row，另一个是内存页管理。我们一个一个来说。

Unsafe Row：二进制数据结构

Unsafe Row 是一种字节数组，它可以用来存储下图所示 Schema 为（userID，name，age，gender）的用户数据条目。总的来说，所有字段都会按照 Schema 中的顺序安放在数组中。其中，定长字段的值会直接安插到字节中，而变长字段会先在 Schema 的相应位置插入偏移地址，再把字段长度和字段值存储到靠后的元素中。更详细的例子我们在第 9 讲说过，你可以去看看。



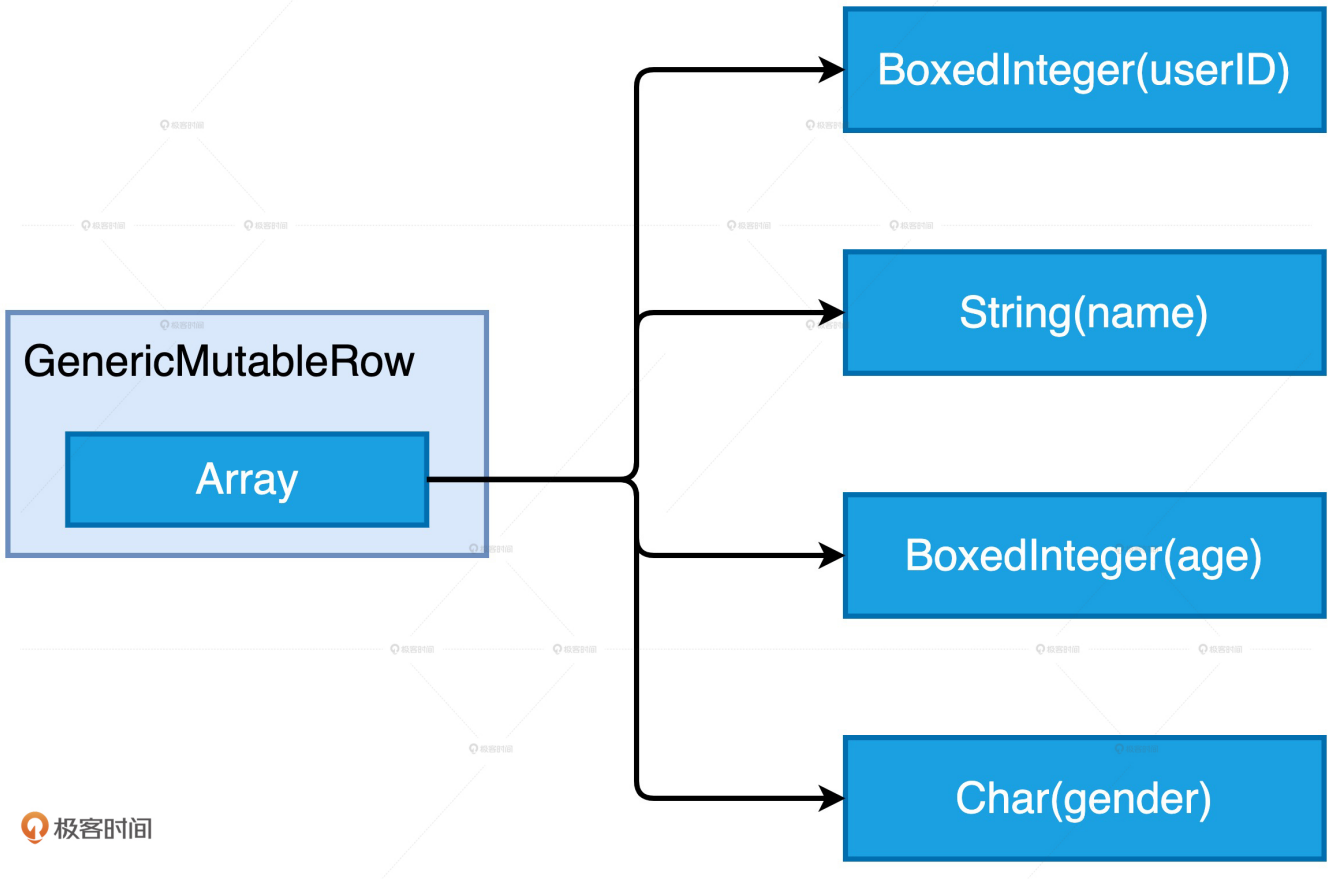
二进制字节数组

那么，这种存储方式有什么优点呢？我们不妨用逆向思维来思考这个问题，如果采用 JVM 传统的对象方式来存储相同 Schema 的数据条目会发生什么。

JVM 至少需要 6 个对象才能存储一条用户数据。其中，GenericMutableRow 用于封装一条数据，Array 用于存储实际的数据值。Array 中的每个元素都是一个对象，如承载整型的 BoxedInteger、承载字符串的 String 等等。这样的存储方式有两个明显的缺点。

首先，存储开销大。我们拿类型是 String 的 name 来举例，如果一个用户的名字叫做“Mike”，它本应该只占用 4 个字节，但在 JVM 的对象存储中，“Mike”会消耗总共 48 个字节，其中包括 12 个字节的对象头信息、8 字节的哈希编码、8 字节的字段值存储和另外 20 个字节的其他开销。从 4 个字节到 48 个字节，存储开销可见一斑。

其次，在 JVM 堆内内存中，对象数越多垃圾回收效率越低。因此，一条数据记录用一个对象来封装是最好的。但是，我们从下图中可以看到，JVM 需要至少 6 个对象才能存储一条数据记录。如果你的样本数是 1 千亿的话，这意味着 JVM 需要管理 6 千亿的对象，GC 的压力就会陡然上升。



JVM存储数据条目

我们反过来再看 UnsafeRow，**字节数组的存储方式在消除存储开销的同时，仅用一个数组对象就能轻松完成一条数据的封装，显著降低 GC 压力**，可以说是一举两得。由此可见，Unsafe Row 带来的潜在性能收益还是相当可观的。不过，Tungsten 并未止步于此，为了统一堆外与堆内内存的管理，同时进一步提升数据存储效率与 GC 效率，Tungsten 还推出了基于内存页的内存管理模式。

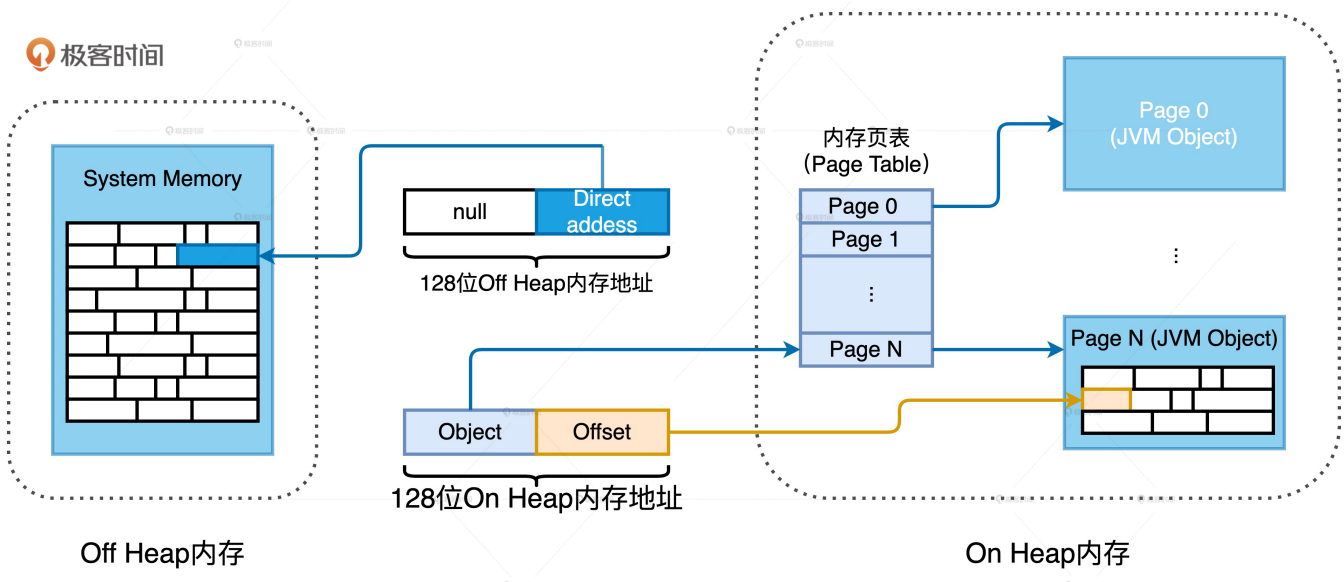
基于内存页的内存管理

为了统一管理 Off Heap 和 On Heap 内存空间，Tungsten 定义了统一的 128 位内存地址，简称 Tungsten 地址。Tungsten 地址分为两部分：前 64 位预留给 Java Object，后 64 位是偏移地址 Offset。但是，同样是 128 位的 Tungsten 地址，Off Heap 和 On Heap 两块内存空间在寻址方式上截然不同。

对于 On Heap 空间的 Tungsten 地址来说，前 64 位存储的是 JVM 堆内对象的引用或者说指针，后 64 位 Offset 存储的是数据在该对象内的偏移地址。而 Off Heap 空间则完全不同，在堆外的空间中，由于 Spark 是通过 Java Unsafe API 直接管理操作系统内存，不存在内存对象的概念，因此前 64 位存储的是 null 值，后 64 位则用于在堆外空间中直接寻址操作系统的内存空间。

显然，在 Tungsten 模式下，管理 On Heap 会比 Off Heap 更加复杂。这是因为，在 On Heap 内存空间寻址堆内数据必须经过两步：第一步，通过前 64 位的 Object 引用来定位 JVM 对象；第二步，结合 Offset 提供的偏移地址在堆内内存空间中找到所需的数据。

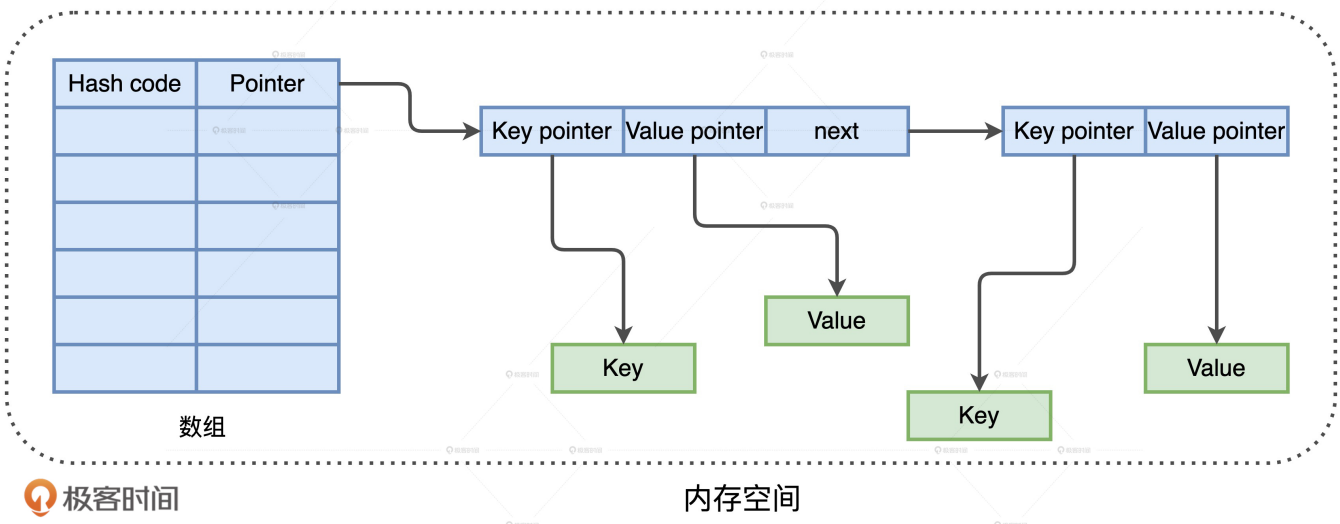
JVM 对象地址与偏移地址的关系，就好比是数组的起始地址与数组元素偏移地址之间的关系。给定起始地址和偏移地址之后，系统就可以迅速地寻址到数据元素。因此，在上面的两个步骤中，如何通过 Object 引用来定位 JVM 对象就是关键了。接下来，我们就重点解释这个环节。



堆外、堆内不同的寻址方式

如上图所示，Tungsten 使用一种叫做页表（Page Table）的数据结构，来记录从 Object 引用到 JVM 对象地址的映射。页表中记录的是一个又一个内存页（Memory Page），内存页实际上就是一个 JVM 对象而已。只要给定 64 位的 Object 引用，Tungsten 就能通过页表轻松拿到 JVM 对象地址，从而完成寻址。

那么，Tungsten 使用这种方式来管理内存有什么收益呢？我们不妨以常用的 HashMap 数据结构为例，来对比 Java 标准库（java.util.HashMap）和 Tungsten 模式下的 HashMap。



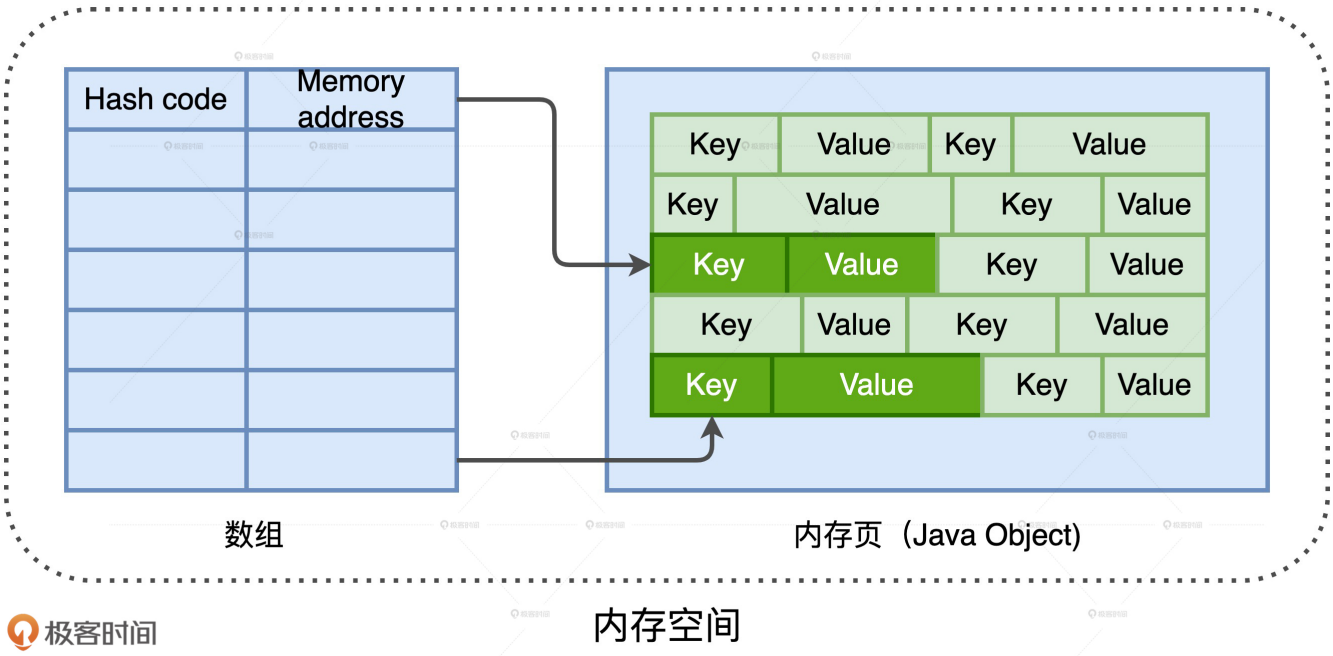
java.util.HashMap

Java 标准库采用数组加链表的方式来实现 HashMap，如上图所示，数组元素存储 Hash code 和链表头。链表节点存储 3 个元素，分别是 Key 引用、Value 引用和下一个元素的地址。一般来说，如果面试官要求你实现一个 HashMap，我们往往也会采用这种实现方式。

但是，这种实现方式会带来两个弊端。

首先是存储开销和 GC 负担比较大。结合上面的示意图我们不难发现，存储数据的对象值只占整个 HashMap 一半的存储空间，另外一半的存储空间用来存储引用和指针，这 50% 的存储开销还是蛮大的。而且我们发现，图中每一个 Key、Value 和链表元素都是 JVM 对象。假设，我们用 HashMap 来存储一百万条数据条目，那么 JVM 对象的数量至少是三百万。由于 JVM 的 GC 效率与对象数量成反比，因此 java.util.HashMap 的实现方式对于 GC 并不友好。

其次，在数据访问的过程中，标准库实现的 HashMap 容易降低 CPU 缓存命中率，进而降低 CPU 利用率。链表这种数据结构的特点是，对写入友好，但访问低效。用链表存储数据的方式确实很灵活，这让 JVM 可以充分利用零散的内存区域，提升内存利用率。但是，在对链表进行全量扫描的时候，这种零散的存储方式会引入大量的随机内存访问（Random Memory Access）。相比顺序访问，随机内存访问会大幅降低 CPU cache 命中率。



Tungsten HashMap

那么，针对以上几个弊端，Tungsten 又是怎么解决的呢？我们从存储开销、GC 效率和 CPU cache 命中率分别来看。

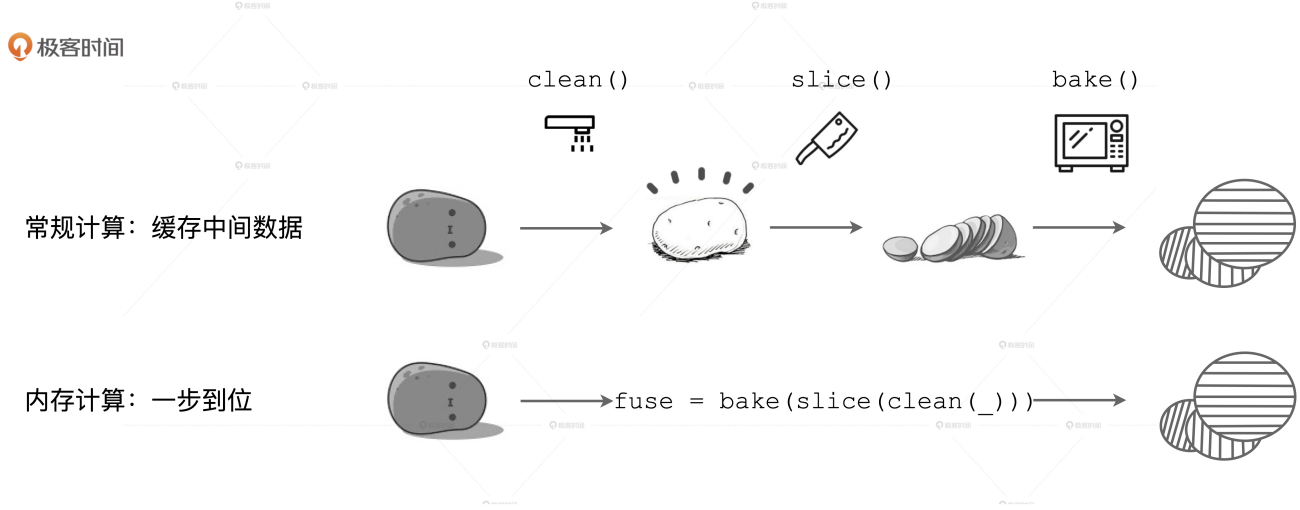
首先，Tungsten 放弃了链表的实现方式，使用数组加内存页的方式来实现 HashMap。数组中存储的元素是 Hash code 和 Tungsten 内存地址，也就是 Object 引用外加 Offset 的 128 位地址。Tungsten HashMap 使用 128 位地址来寻址数据元素，相比 java.util.HashMap 大量的链表指针，在存储开销上更低。

其次，Tungsten HashMap 的存储单元是内存页，内存页本质上是 Java Object，一个内存页可以存储多个数据条目。因此，相比标准库中的 HashMap，使用内存页大幅缩减了存储所需的对象数量。比如说，我们需要存储一百万条数据记录，标准库的 HashMap 至少需要三百万的 JVM 对象才能存下，而 Tungsten HashMap 可能只需要几个或是十几个内存页就能存下。对比下来，它们所需的 JVM 对象数量可以说是天壤之别，显然，Tungsten 的实现方式对于 GC 更加友好。

再者，内存页本质上是 JVM 对象，其内部使用连续空间来存储数据，内存页加偏移量可以精准地定位到每一个数据元素。因此，在需要扫描 HashMap 全量数据的时候，得益于内存页中连续存储的方式，内存的访问方式从原来的随机访问变成了顺序读取（Sequential Access）。顺序内存访问会大幅提升 CPU cache 利用率，减少 CPU 中断，显著提升 CPU 利用率。

如何理解 WSCG?

接下来，我们再说说 WSCG。首先，WSCG 到底是什么？这就要提到内存计算的第二层含义了，它指的是在同一个 Stage 内部，把多个 RDD 的 compute 函数捏合成一个，然后把这一个函数一次性地作用在输入数据上。不过，这种捏合方式采用的是迭代器嵌套的方式。例如，土豆工坊中对于 Stage0 的处理，也就是下图中的 fuse 函数。它仅仅是 clean、slice、bake 三个函数的嵌套，并没有真正融合为一个函数。



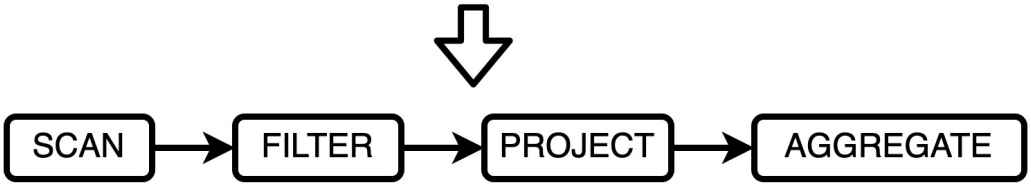
内存计算的第二层含义

WSCG 指的是基于同一 Stage 内操作符之间的调用关系，生成一份“手写代码”，真正把所有计算融合为一个统一的函数。

什么是火山迭代模型?

那么，我们真的有必要把三个函数体融合成一个函数，甚至生成一份“手写代码”吗？迭代器嵌套的函数调用难道还不够吗？坦白说，迭代器嵌套还真不够。原因在于，迭代器嵌套的计算模式会涉及两种操作，**一个是内存数据的随机存取，另一个是虚函数调用（next）**。这两种操作都会降低 CPU 的缓存命中率，影响 CPU 的工作效率。这么说比较抽象，我们来举个小例子。

```
select count(userId) from citizens where city = 'Beijing'
```



SQL查询与语法树

假设，现在有一张市民表，我们要从中统计在北京的人数。对应的语法树非常简单，从左到右，分别是数据扫描、过滤、投影和聚合。语法树先是经过“左脑”Catalyst 优化器转换为 Physical Plan，然后交付执行。Tungsten 出现以前，Spark 在运行时采用火山迭代模型来执行计算。这里，咱们需要先简单地介绍一下火山迭代模型（Volcano Iteration Model，以下简称 VI 模型）。

VI 模型这种计算模式依托 AST 语法树，对所有操作符（如过滤、投影）的计算进行了统一封装，所有操作符都要实现 VI 模型的迭代器抽象。简单来说就是，所有操作符都需要实现 hasNext 和 next 方法。因此，VI 模型非常灵活、扩展能力很强，任何一个算子只要实现了迭代器抽象，都可以加入到语法树当中参与计算。另外，为了方便操作符之间的数据交换，VI 模型对所有操作符的输出也做了统一的封装。

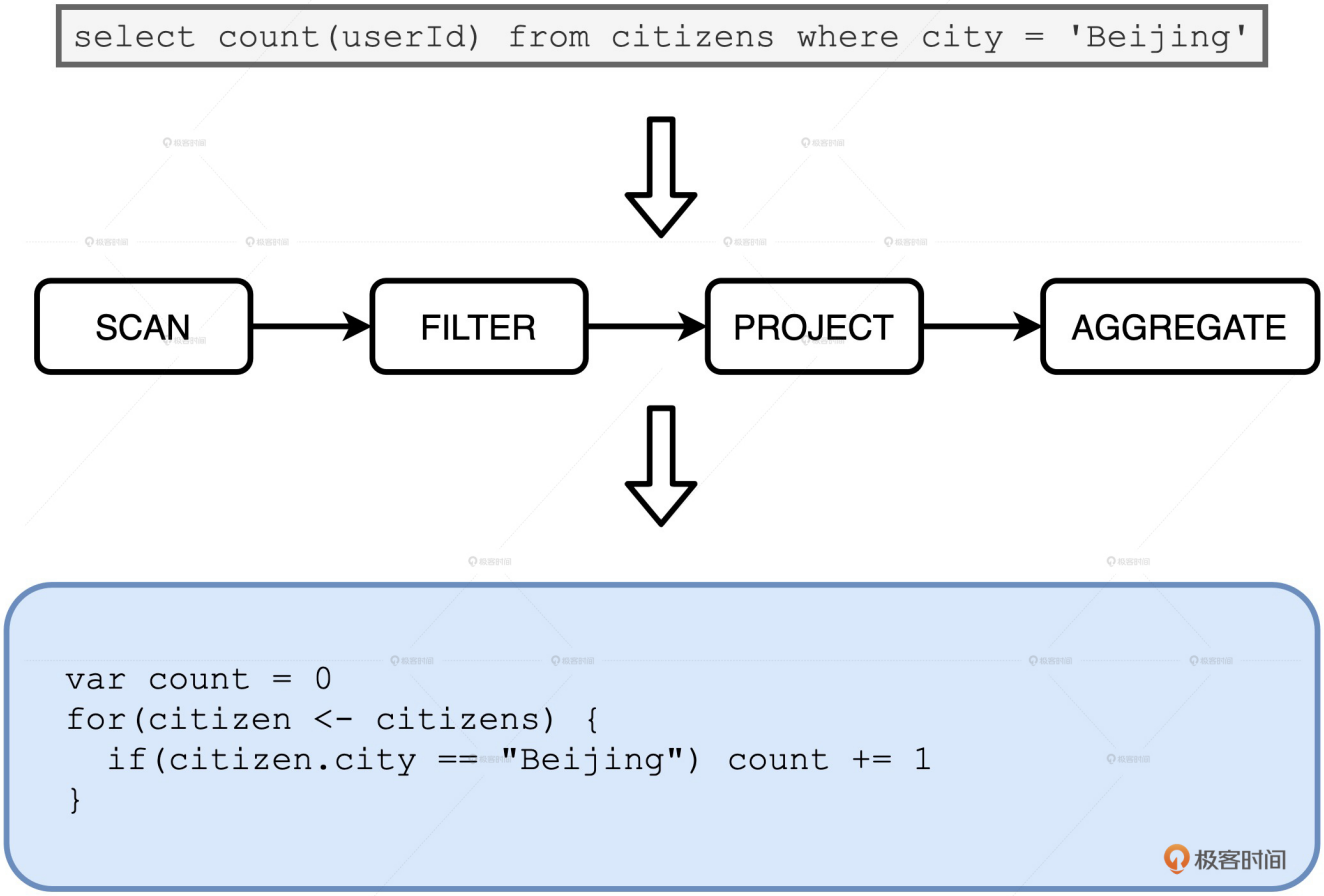
那么，如果上面的查询使用 VI 模型去执行计算的话，都需要经过哪些步骤呢？对于数据源中的每条数据条目，语法树当中的每个操作符都需要完成如下步骤：

1. 从内存中读取父操作符的输出结果作为输入数据
2. 调用 hasNext、next 方法，以操作符逻辑处理数据，如过滤、投影、聚合等等
3. 将处理后的结果以统一的标准形式输出到内存，供下游算子消费

因此，任意两个操作符之间的交互都会涉及我们最开始说的两个步骤，也就是内存数据的随机存取和虚函数调用，而它们正是 CPU 有效利用率低下的始作俑者。

WSCG 的优势是什么?

Tungsten 引入 WSCG 机制，正是为了消除 VI 模型引入的计算开销。这是怎么做到的呢？接下来，咱们还是以市民表的查询为例，先来直观地感受一下 WSCG 的优势。



手写代码示例

对于刚刚的查询语句，WSCG 会结合 AST 语法树中不同算子的调用关系，生成如上图所示的“手写代码”。在这份手写代码中，我们把数据端到端的计算逻辑（过滤、投影、聚合）一次性地进行了实现。

这样一来，我们利用手写代码的实现方式不仅消除了操作符，也消除了操作符的虚函数调用，更没有不同算子之间的数据交换，计算逻辑完全是一次性地应用到数据上。而且，代码中的每一条指令都是明确的，可以顺序加载到 CPU 寄存器，源数据也可以顺序地加载到 CPU 的各级缓存中，从而大幅提升了 CPU 的工作效率。

当然，WSCG 在运行时生成的代码和我们这里举例的手写代码在形式上还有差别。不过，这也并不影响我们对于 WSCG 特性和优势的理解。看到这里，你可能会问：“WSCG 不就是运行时的代码重构吗？”没错，**本质上，WSCG 机制的工作过程就是基于一份“性能较差的代码”，在运行时动态地（On The Fly）重构出一份“性能更好的代码”。**

WSCG 是如何在运行时动态生成代码的？

问题来了，WSCG 是怎么在运行时动态生成代码的呢？

我们还是以刚刚市民表的查询为例，语法树从左到右有 Scan、Filter、Project 和 Aggregate4 个节点。不过，因为 Aggregate 会引入 Shuffle、切割 Stage，所以这 4 个节点会产生两个 Stage。又因为 WSCG 是在一个 Stage 内部生成手写代码，所以，我们把目光集中到前三个操作符 Scan、Filter 和 Project 构成的 Stage。

语法树的第一个Stage

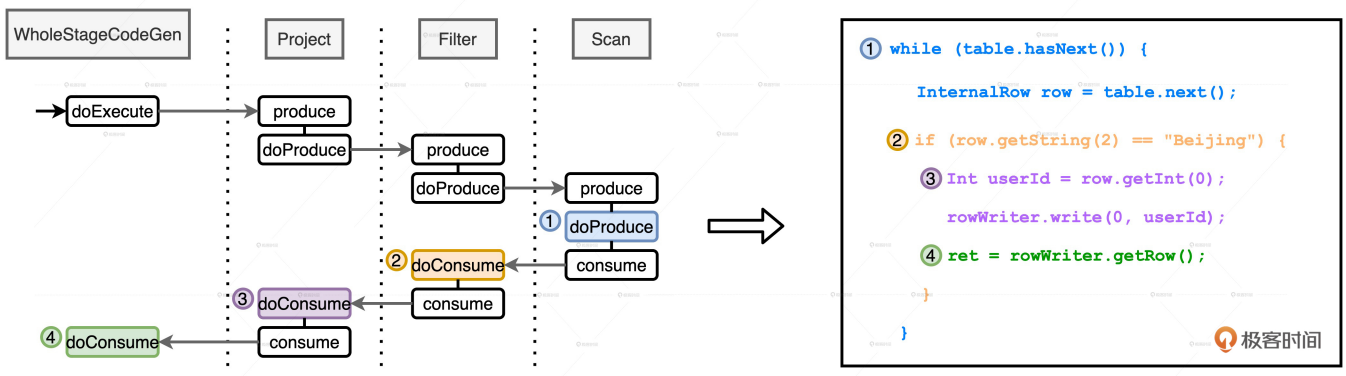
上一讲中我们说了，Spark Plan 在转换成 Physical Plan 之前，会应用一系列的 Preparation Rules。这其中很重要的一环就是 CollapseCodegenStages 规则，它的作用正是尝试为每一个 Stages 生成“手写代码”。

总的来说，手写代码的生成过程分为两个步骤：

从父节点到子节点，递归调用 doProduce，生成代码框架

从子节点到父节点，递归调用 doConsume，向框架填充每一个操作符的运算逻辑

这么说比较抽象，咱们以上面的第一个 Stage 为例，来直观地看看这个代码生成的过程。



WSCG时序图

首先，在 Stage 顶端节点也就是 Project 之上，添加 WholeStageCodeGen 节点。WholeStageCodeGen 节点通过调用 doExecute 来触发整个代码生成过程的计算。doExecute 会递归调用子节点的 doProduce 函数，直到遇到 Shuffle Boundary 为止。这里，Shuffle Boundary 指的是 Shuffle 边界，要么是数据源，要么是上一个 Stage 的输

出。在叶子节点（也就是 Scan）调用的 doProduce 函数会先把手写代码的框架生成出来，如图中右侧蓝色部分的代码。

然后，Scan 中的 doProduce 会反向递归调用每个父节点的 doConsume 函数。不同操作符在执行 doConsume 函数的过程中，会把关系表达式转化成 Java 代码，然后把这份代码像做“完形填空”一样，嵌入到刚刚的代码框架里。比如图中橘黄色的 doConsume 生成的 if 语句，其中包含了判断地区是否为北京的条件，以及紫色的 doConsume 生成了获取必需字段 userId 的 Java 代码。

就这样，Tungsten 利用 CollapseCodegenStages 规则，经过两层递归调用把 Catalyst 输出的 Spark Plan 加工成了一份“手写代码”，并把这份手写代码会交付给 DAGScheduler。拿到代码之后，DAGScheduler 再去协调自己的两个小弟 TaskScheduler 和 SchedulerBackend，完成分布式任务调度。

小结

Tungsten 是 Spark SQL 的“右脑”，掌握它的特性和优势对 SparkSQL 的性能调优来说至关重要。具体来说，我们可以从它对内核引擎的两方面改进入手：数据结构设计和 WSCG。

在数据结构方面，我们要掌握 Tungsten 的两项改进。

首先，Tungsten 设计了 UnsafeRow 二进制字节序列来取代 JVM 对象的存储方式。这不仅可以提升 CPU 的存储效率，还能减少存储数据记录所需的对象个数，从而改善 GC 效率。

其次，为了统一管理堆内与堆外内存，Tungsten 设计了 128 位的内存地址，其中前 64 位存储 Object 引用，后 64 位为偏移地址。

在堆内内存的管理上，基于 Tungsten 内存地址和内存页的设计机制，相比标准库，Tungsten 实现的数据结构（如 HashMap）使用连续空间来存储数据条目，连续内存访问有利于提升 CPU 缓存命中率，从而提升 CPU 工作效率。由于内存页本质上是 Java Object，内存页管理机制往往能够大幅削减存储数据所需的对象数量，因此对 GC 非常友好的。

对于 Tungsten 的 WSCG，我们要掌握它的概念和优势。

首先，WSCG 指的是基于同一 Stage 内操作符之间的调用关系，生成一份“手写代码”，来把所有计算融合为一个统一的函数。本质上，WSCG 机制的工作过程，就是基于一份“性能较差的代码”，在运行时动态地重构出一份“性能更好的代码”。

更重要的是，“手写代码”解决了 VI 计算模型的两个核心痛点：操作符之间频繁的虚函数调用，以及操作符之间数据交换引入的内存随机访问。手写代码中的每一条指令都是明确的，可以顺序加载到 CPU 寄存器，源数据也可以顺序地加载到 CPU 的各级缓存中，因此，CPU 的缓存命中率和工作效率都会得到大幅提升。

每日一练

1. 针对排序操作，你认为 Tungsten 在数据结构方面有哪些改进呢？
2. 你认为表达式代码生成（Expression Codegen）和全阶段代码生成（Whole Stage Codegen）有什么区别和联系呢？

期待在留言区看到你的思考和答案，我们下一讲见！

提建议

更多课程推荐

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 22 | Catalyst物理计划：你的SQL语句是怎么被优化的（下）？

精选留言 (4)

写留言

aof
2021-05-05

花了一个五一的时间，从第一篇追上了老师的更新进度哈哈！老师厉害，学到了很多，尤其Spark SQL这块，还是要对着老师讲的东西，再去认真的读下源码才能有更深的理解

展开

作者回复: 老弟V5，进度神速~

Fendora范东_
2021-05-05

有个地方没理解

1.onheap内存寻址所说的内存页表，保存的是对象引用到jvm对象地址的映射，那它应该也是一个map结构，它是不是用下面所说的Tungsten.hashmap实现的?
2.我理解不是，下面说的Tungsten.hashmap实例，我理解它是一个完整查询结构:要查某条数据，先计算key的hashcode，然后拿到128位内存地址，然后高64位(堆内对象引用...
展开

1

zxk
2021-05-05

问题二：Spark SQL 解析为语法树后，在不使用 Expression Codegen 的情况下，表达式节点每次执行都需要进行 Spark 内部的一些相关操作（如做一些操作类型匹配），那么 Spark 自身机制的开销可能大于我们需要执行的计算的开销，因此需要 Expression Codegen 对表达式进行代码生成，此时侧重于对表达式自身的优化；而 WSCG 则侧重与多个函数之间的合并，两者侧重点并不相同。...
展开

斯盖丸
2021-05-05

老师，请问On heap寻址里的Object引用和偏移地址分别对应的是什么？Object引用是一条Row，偏移地址里是Row的一个字段或者说是列吗?
展开